An Example of Synchronous Design of Embedded Real-Time Systems based on IMA*

Abdoulaye Gamatié, Thierry Gautier and Paul Le Guernic

IRISA / University of Rennes 1 / INRIA, F-35042 RENNES, France.
{*abdoulaye.gamatie, thierry.gautier, paul.leguernic*}*@irisa.fr*

Abstract. We present a study on the design of embedded real-time systems in general, and avionic systems in particular. The synchronous language SIGNAL is used to describe a real world avionic application based on the recent Integrated Modular Avionics concept (IMA). The exposed case study shows how the synchronous technology helps for a reliable and modular design of real-time systems. One major advantage of this technology is the availability of formal tools and techniques for verification and validation, which are important for safety-critical systems such as avionics systems.

1 Introduction

Two noteworthy observations about embedded systems in today's technologies are their ubiquitousness and pervasiveness. The concerned application domains include, among others, avionics, automotive and railway systems, industrial automation and process control, robots, and medical technology. In all these domains, embedded systems play a critical role, since a failure can put human lives at stake or have at least serious economical consequences. Thus, we understand how much it is crucial to be able to guarantee *a priori* the correctness of these systems (i.e. to ensure that such a system behaves with respect to its requirements). Embedded systems consist in combinations of digital and analog components, designed to provide dedicated functions. They are often characterized by real-time constraints. Thus, their correctness depends both on the logical results and on the instants at which these results are produced. Among challenges in the design of embedded real-time systems, are cost-effectiveness, time to market, development effort, as well as correctness and reliability of the implementation. Broad discussions of these challenges can be found in the literature [21]. Well-established design frameworks offer very promising ways to deal with these issues. Such frameworks must provide at least a means to describe a system without ambiguity, to check desired properties of systems, and to automatically generate code with respect to requirements.

* This work has been partially supported by the European project IST SAFEAIR (*Advanced Design Tools for Aircraft Systems and Airborne Software* - `http://www.safeair.org`).

The application of formal methods is now widely accepted for the development of safety critical systems such as embedded real-time systems [18]. This success is due not only to the high degree of confidence they provide in designs, but also to the availability of several practical tools accompanying the proposed methods. Meanwhile, *model-based* approaches have gained prominence in system design in recent years. As a matter of fact, modeling is becoming increasingly essential to the design activity for embedded systems. It allows experiments without necessarily having a physical implementation of the system, thus enabling a high flexibility in design choices and earlier decisions. Besides that, strong points emphasized in [21] are genericity, abstraction, and formal techniques for analysis and predictability. Finally, *component-based* approaches provide a way to significantly reduce overall development costs through modularity and re-usability.

The synchronous technology [4] has emerged as a very efficient and practical solution to the need for a safe design of embedded systems. The underlying theory of this technology is that of discrete event systems and automata theory. Time is *logical*: it is handled according to partial order and simultaneity of events. Durations of execution are viewed as constraints to be verified at the implementation level. Typical synchronous languages are ESTEREL, LUSTRE and SIGNAL. They mainly differ from each other in their programming style. ESTEREL adopts an imperative style, whereas the two others are data-flow oriented (LUSTRE is functional and SIGNAL is relational). Examples of associated design environments are SCADE[1] and RT-BUILDER[2] (SILDEX), which are currently used in aircraft industries (e.g. *Airbus* and *Snecma*).

In this paper, we first introduce the *Integrated Modular Avionics* concept (IMA) and the APEX – ARINC 653 standard (Section 2). Then, we give an overview of the synchronous language SIGNAL in section 3. A case study is presented in Section 4. It includes a SIGNAL description of a real world avionic application based on IMA. It also illustrates how timing issues are addressed using the SIGNAL language. In Section 5, we discuss our approach and mention some relevant related work. Finally, conclusions are given in Section 6.

2 The integrated modular avionics concept

Avionic systems are a good example of safety critical systems. Failures in such systems lead to heavy consequences ranging from loss of money to loss of life. Within these systems, components are usually associated with criticality levels that allow to identify the most critical components and prevent them from being jeopardized during their execution by less critical ones. Therefore, *federated* architectures [19] have appeared as a natural solution to the efficient design of avionic systems: components with different criticality levels are separately supported by dedicated hardware. A great advantage of that architecture is fault containment. However, there exists a potential risk of massive usage of resources since each component in the system may require its dedicated hardware. As a

[1] `http://www.esterel-technologies.com`.
[2] `http://www.tni-software.com`.

result, one can mention the rise of maintenance costs. The introduction of *Integrated Modular Avionics* (IMA) [1] aims to propose solutions to the obstacles encountered in federated architectures.

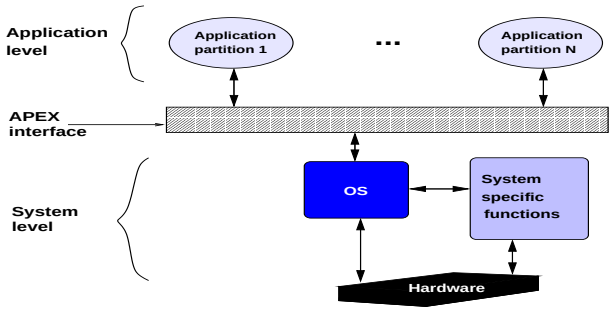


Fig. 1. APEX – ARINC 653 interface.

IMA allows several components with possibly different criticality to share the same hardware resources. It ensures a safe allocation of shared resources so that no fault propagation occurs from one component to another component. This is achieved through *partitioning* of resources with respect to available time and memory capacities. A *partition* is a logical allocation unit resulting from a functional decomposition of the system. Partitioning promotes verification, validation, and certification. IMA platforms consist of a number of *modules* grouped in cabinets throughout the aircraft. A module can contain several partitions that possibly belong to applications of different criticality levels. Mechanisms are provided in order to prevent a partition from having "abnormal" access to the memory area of another partition. The processor is allocated to each partition for a fixed time window within a major time frame maintained by the module level operating system (OS). A partition cannot be distributed over multiple processors either in the same module or in different modules. Finally, partitions communicate asynchronously via logical *ports* and *channels*. Message exchanges rely on two transfer modes: *sampling* mode and *queuing* mode. In the former, no message queue is allowed. A message remains in the source port until it is transmitted via the channel or it is overwritten by a new occurrence of the message. A received message remains in the destination port until it is overwritten. A refresh period attribute is associated with each sampling port. When reading a port, a *validity* parameter indicates whether the age of the read message is consistent with the required refresh period attribute of the port. In the queuing mode, ports are allowed to store messages from a source partition in queues until they are received by the destination partition. Here, the queuing discipline for messages is First-In First-Out (FIFO).

Partitions are composed of *processes* that represent the executive units[3]. Processes run concurrently and execute functions associated with the partition in which they are contained. Each process is uniquely characterized by information (like its period, priority, or deadline time) useful to the partition level OS which is responsible for the correct execution of processes within a partition. The scheduling policy for processes is priority preemptive. Communications between processes are achieved by three basic mechanisms. The bounded *buffer* allows to send and receive messages following a FIFO policy. The *event* permits the application to notify processes of the occurrence of a condition for which they may be waiting. The *blackboard* is used to display and read messages: no message queues are allowed, and any message written on a blackboard remains there until the message is either cleared or overwritten by a new instance of the message. Synchronizations are achieved using a *semaphore*.

Several standards for software and hardware have been defined for IMA. Here, we particularly consider the APEX – ARINC 653 standard [2], which proposes an OS interface for IMA applications, called *Avionics Application Software Standard Interface* (cf. Figure 1). It includes, among others, services for communication between partitions on the one hand and between processes on the other hand, synchronization services for processes, and partition and process management services.

3 The synchronous language SIGNAL

The SIGNAL language [13] handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as `x` in the language, and implicitly indexed by discrete time (denoted by t in the semantic notation). At a given instant, a signal may be present, at which point it holds a value; or absent, at which point it is denoted by the special symbol $\perp$ in the semantic notation. There is a particular type of signals called `event`. A signal of this type is always *true* when it is present (otherwise, it is $\perp$). The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x` (which is of type `event`) in the language. Signals that have the same clock are said to be *synchronous*. A SIGNAL *process* is a system of equations over signals, and a *program* is a process. SIGNAL relies on a handful of primitive constructs which are combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. In the following, we give a sketch of primitive constructs (bold-faced) and few derived constructs of SIGNAL (italics). For each one, we respectively mention its corresponding syntax and definition:

Functions/Relations: `y:= f(x1,...,xn)` $\stackrel{def}{=} y_t \neq\perp\Leftrightarrow x1_t \neq\perp\Leftrightarrow \ldots \Leftrightarrow xn_t \neq\perp$, $\forall t$: $y_t = f(x1_t, \ldots, xn_t)$.

Delay: `y:= x $ 1 init y0` $\stackrel{def}{=} x_t \neq\perp\Leftrightarrow y_t \neq\perp$, $\forall t > 0$: $y_t = x_{t-1}, y_0 = y0$.

Down sampling: `y:= x when b` $\stackrel{def}{=} y_t = x_t$ if $b_t = true$, else $y_t = \perp$. The derived

[3] An ARINC partition/process is akin a UNIX process/task.

statement `y:= when b` is equivalent to `y:= b when b`.
Deterministic merging: `z:= x default y` $\stackrel{def}{=} z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
Parallel composition: `(| P | Q |)` $\stackrel{def}{=}$ union of equations associated with `P` and `Q`.
Hiding: `P where x` $\stackrel{def}{=}$ `x` is local to the process `P`.
Clock extraction: `h:= ^x` $\stackrel{def}{=}$ `h := (x = x)`.
Synchronizing: `x1 ^= x2` $\stackrel{def}{=}$ `(| h := (^x1 = ^x2) |) where h`.
Clock union: `h:= x1 ^+ x2` $\stackrel{def}{=}$ `h:= ^x1 default ^x2`.
Memory: `y:= x cell b init y0` $\stackrel{def}{=}$ `(| y := x default (y$1 init y0) | (y ^= x ^+ (when b) |)`.
Context-dependent memory: `y:= (var x) init c` $\stackrel{def}{=}$ `y` gets values of `x` at the context clock (`c` is its initial value).

The SIGNAL language also provides a process frame in which any process may be "encapsulated". This allows to abstract a process to an interface, so that the process can be used afterwards as a black box through its interface which describes the input-output signals and parameters. The process frame enables the definition of sub-processes. Sub-processes that are only specified by an interface without internal behavior are considered as external (they may be separately compiled processes or physical components). On the other hand, SIGNAL allows to import external modules (e.g. C++ functions). Finally, put together, all these features of the language favor modularity and re-usability. The mathematical foundations of SIGNAL enable formal verification and analysis techniques. We can distinguish two kinds of properties: *functional* and *non functional* properties.

Functional properties consist of *invariant properties* on the one hand (e.g. a program exhibits no contradiction between clocks of involved signals), and *dynamic properties* on the other hand (e.g. reachability, liveness). The SIGNAL compiler itself addresses only invariant properties (here, the compiler includes more functionalities than classical compilers. For example, in addition to habitual syntax or type checking, it implements some static verification algorithms and allows for automatic generation of an optimized code). For a given SIGNAL program, the compiler checks the consistency of constraints between clocks of signals, and statically proves properties (e.g. determinism, absence of cyclic definitions, absence of empty clocks to ensure a consistent reactivity of the program). A major part of the compiler task is referred to as the *clock calculus*. Dynamic properties are addressed using other tools like SIGALI [16], a formal system that can be used for model checking.

Non functional properties include *temporal properties*, which are of high interest for real-time systems. A technique has been defined in order to allow timing analysis of SIGNAL programs [12]. Basically, it consists of formal transformations of a program initially describing an application, which yield another SIGNAL program that corresponds to a *temporal interpretation* of the initial one. The new program will serve as an *observer* of the initial program. An observer of a program `P` is an *abstraction* $\mathcal{O}$(`P`) of `P` in which we specify only the properties we want to check. Here, the term "abstraction" means that $\mathcal{O}$(`P`) does not constrain the original behavior of `P` whenever the two programs get composed. As

shown in Figure 2, the observer receives from the observed program the signals required for a given analysis and indicates whether or not the considered properties have been satisfied (e.g. this can be expressed through output boolean signals as for LUSTRE programs [11]). The use of observers for verification is very practical because they can be easily described in the same formalism as the observed program. Thus, there is no need to combine different formalisms as in other analysis techniques like some model-checking techniques, which associate temporal logics with automata.

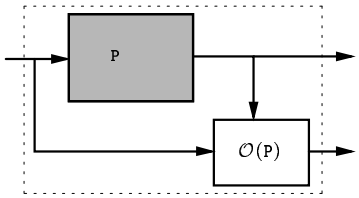


Fig. 2. Analysis of a program using an observer program (P is the observed program and $\mathcal{O}$(P) the observer.

The development environment associated with SIGNAL is called POLYCHRONY[4]. In the next section, we will show how the co-simulation of the temporal interpretation of a SIGNAL program (i.e. the observer) together with the program (i.e. the observed one) provides timing information like latencies, allowing, for instance, to compute worst case execution times within POLYCHRONY.

4 IMA-based modeling: a case study

In this section, we will first illustrate the description of an avionic application. Then, we will address its temporal evaluation. The presented application is a real world case study from the avionic industry. For reasons of confidentiality, we have deliberately modified the names of entities. However, it does not hinder the comprehension of the exposed approach.

4.1 Informal description

The application we are considering is called SATMAINT. It calculates and emits to other components of the global system some general parameters and information corresponding to the maintenance phase (this is done cyclically). The partition also acquires and treats maintenance messages received from other components during the execution of the system. SATMAINT communicates with an external device in order to allow an operator to visualize the maintenance information stored in the memory by the application. Finally, the partition periodically checks the availability of maintenance data, and emits a report message.

[4] `http://www.irisa.fr/espresso/Polychrony`.

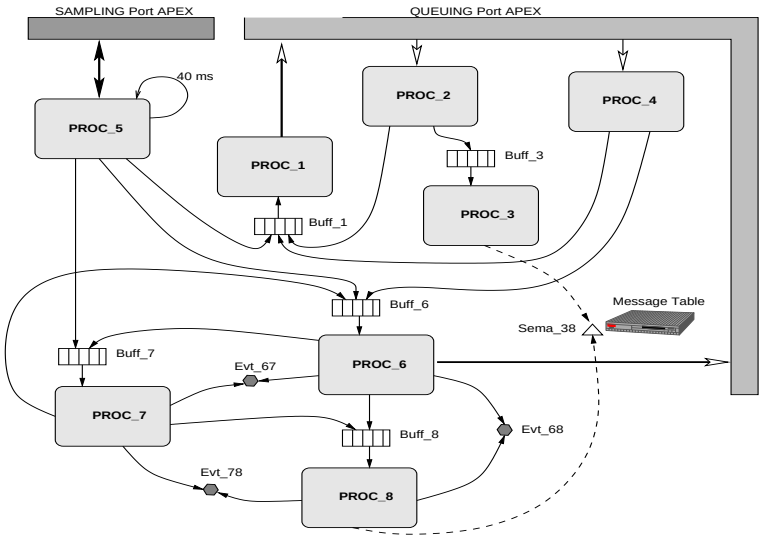


Fig. 3. Informal architecture of the partition SATMAINT.

SATMAINT is represented by a single IMA partition. Its decomposition into processes, as well as the interactions between the processes, are clearly identified in the initial informal specification. This is depicted by Figure 3. There are eight processes (denoted by *Proc_i* where $i \in \{1..8\}$). A major part of the function affected to the partition is achieved by *Proc_6*, *Proc_7* and *Proc_8*. The other processes are rather in charge of collecting required data in order to treat failure messages, and calculate maintenance information that are either saved or sent to the operator.

The processes communicate within the partition using five buffers, noted *buff_i* where i is the identifier of the process that receives its messages from the buffer. In addition, three events in the partition are used for communication. Each one is represented as *evt_ij*, where i and j indicate the two processes that communicate via this event. Finally, the unique semaphore used here is *sema_38*. It allows *Proc_3* and *Proc_8* to accede to *Message_table* in mutual exclusion. The partition SATMAINT communicates with its environment via sampling ports and queuing ports.

4.2 The SIGNAL model definition

Based on the informal description of the application seen above, we can now define the corresponding SIGNAL model. For that, we use the components previously presented in [9]. They include SIGNAL models of APEX – ARINC 653 services: inter-process communication and synchronization services, inter-partition communication services, process and partition management services, and time management services. They also include complementary services (which are not

part of APEX) to allow more complete descriptions of applications (e.g. a service for process rescheduling). Finally, generic models of the executive entities (partitions and processes) have been defined. The combination of all these components allows to describe real-time applications as it is illustrated in the following.

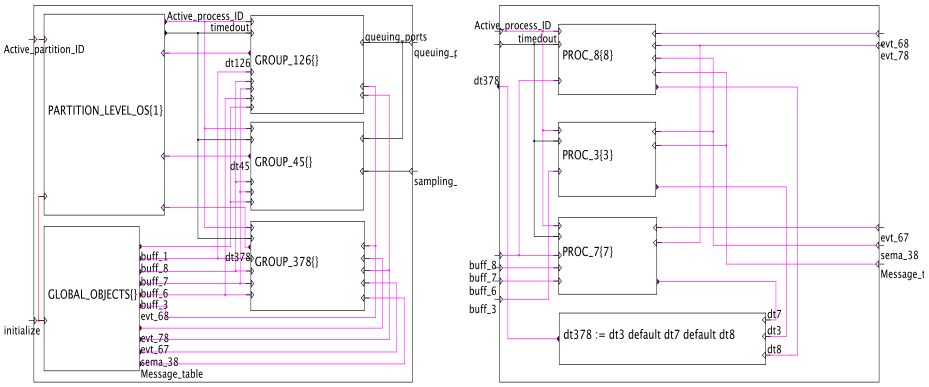


Fig. 4. SIGNAL model of the partition SATMAINT (left), and a zoom in `GROUP_378` (right).

Global view of the partition model. The model of the partition SATMAINT is shown in Figure 4. One can distinguish the component representing the partition-level OS. The box containing the call `GLOBAL_OBJECTS` has been added for structuring. It provides the processes with communication and synchronization mechanisms, and other local resources (e.g. `Message_table`). Processes are grouped into subsets identified by the prefix "`GROUP_`". For instance, `GROUP_378` contains `PROC_3`, `PROC_7` and `PROC_8` as shown in the zoom on Figure 4 (right). All the objects and processes are created at the initialization phase of the partition (denoted by the occurrence of the input signal `initialize`). The input `Active_partition_ID` represents the identifier of the running partition selected by the module-level OS, and it denotes an execution order when it identifies the current partition (the activation of each partition depends on this signal. It is produced by the module-level OS, which is in charge of the management of partitions in a module). Whenever the partition executes, the `PARTITION_LEVEL_OS` selects an active process within the partition. This is represented by its output signal `Active_process_ID`, which is sent to each process. A time information `dt`k is then returned by processes to the partition-level OS. It represents the duration of the current "block" of actions executed by an active process. The signal `timedout` produced by the partition-level OS carries information about the current status of the time counters used within the partition. For instance, a time counter is used for a wait when a process gets interrupted on a service request with time-out. As the partition-level OS is responsible for the man-

agement of time counters, it notifies each interrupted process of the partition with the expiration of its associated time counter. This is reflected by the signal `timedout`.

The partition-level OS model. Figure 5 gives a partial view of the SIGNAL description of the partition-level OS. Its main task is to control the concurrent execution of processes within the partition. Its description requires APEX services (for instance on Figure 5, `CREATE_PROCESS` and `START` allow respectively to create and start processes) and implementation-dependent functions like those which define the scheduling policy (for instance, it is the case of the `PROCESS_SCHEDULINGREQUEST` call in Figure 5).

```
(| (att0,...,att8) := GET_PROCESSES_ATTRIBUTES{}(when initialize)                    (a)
 | (pid0,return_code0) := CREATE_PROCESS{}(att0 when initialize)
 | ...
 | (pid8,return_code8) := CREATE_PROCESS{}(att8 when initialize)                     (b)
 | return_code9 := SET_PARTITION_MODE{}(#NORMAL when initialize)                     (c)
 | return_code10 := START{}(pid0)
 | ...
 | return_code18 := START{}(pid8)                                                   (d)
 | is_running := when (Active_partition_ID = Partition_ID)                          (e)
 | diagnostic := PROCESS_SCHEDULINGREQUEST{}(is_running)                            (f)
 | (Active_process_ID,status,valid) := PROCESS_GETACTIVE{}(is_running)              (g)
 | timedout := UPDATE_COUNTERS{}(dt126 default dt45 default dt378)                  (h)
 | ...
 | switch_on_idle := (when ((Active_partition_ID$1)=Partition_ID))
                      when (not (Active_partition_ID=(Active_partition_ID$1)))     (i)
 | switch_on_normal := (when (not ((Active_partition_ID$1)=Partition_ID)))
                        when (Active_partition_ID=(Active_partition_ID$1))         (j)
 | return_code19 := SET_PARTITION_MODE{}((#IDLE when switch_on_idle)
                    default (#NORMAL when switch_on_normal))                       (k)
 |)
```

Fig. 5. The partition-level OS model.

On the presence of the signal `initialize`, which corresponds to the initialization phase of the partition, process attributes are defined in equation `(a)` (examples of attributes are *priority*, *period*). These attributes are used to create processes and then to start them. Creation does not imply dynamic memory allocation [2]; it only creates a link between the given name and a statically allocated process with a suitable stack area, having the same name. The `START` service only puts the specified process in the "ready" state. For instance, lines `(b)` and `(d)` correspond to the creation and initiation of the process *Proc_8* (identified by `pid8`). The partition is set to the *NORMAL* mode[5] in the equation `(c)`.

[5] There are four operating modes [2]: *IDLE*, *NORMAL*, *COLD_START* and *WARM_START*. For instance, in the *IDLE* mode, the partition does not execute any

The signal `Active_partition_ID` represents the identifier of the running partition (selected by the module-level OS) and it denotes an execution order when it identifies the current partition: this is the meaning of the signal `is_running` (line `(e)`). Therefore, process rescheduling is performed (line `(f)`). Process rescheduling occurs at each activation of the partition (see the sequence diagram illustrated on Figure 10). It also occurs during executions of APEX services that can induce the interrupt of the calling processes. The process with the highest priority in the ready state (identified by `Active_process_ID` at line `(g)`) is designated to be active. On the other hand, time counters are updated (line `(h)`). The signal `timedout` notifies processes with expiration of counters with which they are concerned. The partition continues to execute in that way whenever it is identified by the input signal `Active_partition_ID`.

The partition becomes inactive when the signal `Active_partition_ID` is different from its identifier. This is expressed in the equation `(i)`: the signal of type event, `switch_on_idle`, occurs when the previous value of the input `Active_partition_ID` was equal to `Partition_ID` (the identifier of the current partition), and different from its current value. Similarly, `switch_on_normal` expresses a re-activation of the partition (line `(j)`). The correct mode setting is done in the equation `(k)`.

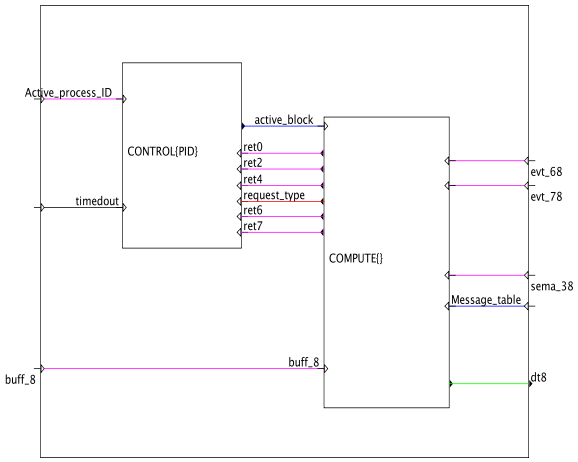


Fig. 6. A SIGNAL model of *Proc_8*.

Processes model. To illustrate how processes are modeled, we focus on *Proc_8*, which belongs to `GROUP_378` on Figure 4. Models of the other processes of SATMAINT are defined in the same way. A well-known modular design principle for

process, whereas in the *NORMAL* mode, the scheduler gets activated (all the required resources in the partition must have been already created).

a system consists in considering separately its *control* and *computation* parts (for instance, this idea has gained a great popularity in hardware design). The model we propose for processes relies on the same principle. Two basic sub-components are distinguished, as shown in Figure 6: *CONTROL* and *COMPUTE*. The former specifies the execution flow of the process. Typically, it is a finite state machine that indicates which statements (or actions) should be executed whenever the process is active (cf. Figure 9). The latter describes the executed statements, which are grouped into *blocks*. Each block is associated with a state of the automaton specified in *CONTROL*. In addition, a block is assumed to be executed without interruption, within a bounded amount of time. To make an analogy, the way the two subparts of the process model interact is similar to what happens in a mode-automaton [15].

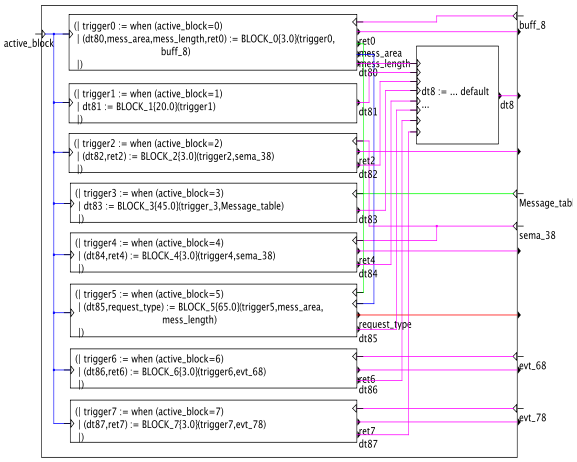
```
scan the buffer buff_8 and retrieve arriving messages;
if a message is retrieved then
   do actions A1;
   perform a "wait_semaphore" on the semaphore sema_38;
   do actions A2 (which require access to Message_Table);
   perform a "signal_semaphore" on the semaphore sema_38;
   do actions A3;
   if the received message denotes an operator request then
      perform a "set_event" on the APEX event evt_68;
   else if the received message denotes another request then
      perform a "set_event" on the APEX event evt_78;
   end if
end if
```

Fig. 7. Informal specification of *Proc_8*.

The informal specification of *Proc_8* is given on Figure 7. The corresponding SIGNAL model is shown on Figures 6, 8 and 9. In the model depicted by Figure 6, the signal `active_block` identifies a block selected in CONTROL. This block is executed *instantaneously*. Of course, one must be careful with the kinds of statements that can be combined in a block. Two sorts of statements can be distinguished: those which may cause an interruption of the running process (e.g. a *RECEIVE_BUFFER* request on an empty buffer), termed *system calls* (in reference to the fact that they involve the partition-level OS); and those that never interrupt a running process (typically data computation functions), referred to as *functions*. Since a block is supposed to be non-interruptible, we impose that it contains either one single system call or one or more functions. This way, the instantaneousness of the block execution is guaranteed to be coherent with its non-interruptibility.

The COMPUTE subpart of *Proc_8* is depicted by Figure 8 (left). Blocks are represented by inner boxes. The statements associated with a block j are

executed whenever the current state of the automaton specified in CONTROL is *blockj* (cf. Figure 9), i.e., whenever the event `triggerj` is present.




```
process BLOCK_0 =
 { SystemTime_type delay; }
 (? event trigger0;
    Comm_ComponentID_type buff_8;
  ! SystemTime_type dt80;
    MessageArea_type mess_area;
    MessageSise_type mess_length;
    ReturnCode_type ret0;
 )
(| (mess_area, mess_length, ret0):=
     RECEIVE_BUFFER{8}((var buff_8)
        when trigger0, 500.0)
 | dt80 := delay when trigger0
 |)

process BLOCK_1 =
 { SystemTime_type delay; }
 (? event trigger1;
  ! SystemTime_type dt81;
 )
(| Actions1{}(when trigger1)
 | dt81 := delay when trigger1
 |)
...
```

Fig. 8. The COMPUTE subpart for *Proc_8*.

For instance, on Figure 8 (right), the first block executed by the process (represented by `BLOCK_0`) contains the system call `RECEIVE_BUFFER` in order to receive a message from `buff_8`. This APEX service takes as input parameters the identifier of the specified buffer and a time-out value (500.0 time units) to wait for a message when the buffer is empty. Output parameters are the message address and size and a return code that reflects the diagnostic of the request. The return code is useful for the CONTROL subpart in order to perform the correct transitions in the automaton. The execution time corresponding to this block, denoted by `dt80`, is defined by the value of the parameter `delay`, which is supposed to have already been calculated off-line (e.g. it could be the worst case execution time of the called APEX service). This information can be obtained by considering a simulation of the associated temporal interpretation as discussed in section 3.

The second block executed by the process (represented by `BLOCK_1`) only contains actions (*A1* in the informal specification) that are not system calls. Their associated execution time is given by `dt81`. After each execution of a block, the corresponding duration is sent to the partition-level OS in order to update time counters. The output signal `dt8` of the COMPUTE subpart (Figure 8, left) first takes the value of any `dt8`k ($k \in \{0,..,7\}$), which is present. It is used in the definition of the output signal `dt378` (which represents the duration of the current block executed by the active process) in `GROUP_378` (cf. Figure 4, right). Finally, the partition-level OS receives `dt378`.

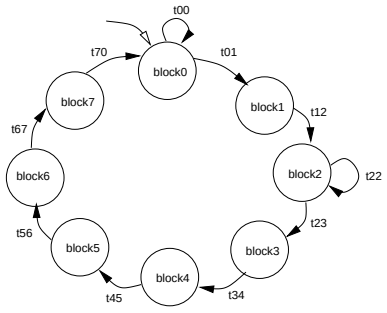


Fig. 9. The automaton associated with the CONTROL subpart for *Proc_8*.

As we can observe, blocks are computed sequentially (from top to bottom) as represented by the transitions labeled by t_{ij} $(i \neq j)$ in the automaton depicted by Figure 9. However, consecutive executions of a same block can sometimes occur. This happens when a system call is executed and the required resource is not yet available. For example, consider the `RECEIVE_BUFFER` request. If there is no message currently in the buffer, the calling process will get suspended on this block. After a message has been sent, the state of the process is set to "ready". As soon as this process becomes active, it re-executes the same block (which induced its suspension) to retrieve the message that was sent. Such transitions are labeled t_{ii} in the automaton.

The UML sequence diagram[6] depicted by Figure 10, called *execution*, illustrates how the partition-level OS interacts with a process during the execution of the partition.

After the initialization phase, the partition gets activated (i.e. when receiving *Active_partition_ID*). The partition-level OS selects an active process within the partition. Then, the CONTROL subpart of each process checks whether or not the concerned process can execute. In the diagram, this is denoted by the *optional* action (represented by a box labeled *opt*). In the case a process is designated by the OS, this action is performed: the process executes a block from its COMPUTE subpart, and the duration corresponding to the executed block is returned to the partition-level OS in order to update time counters. The execution of the model of the partition follows this basic pattern until the module-level OS selects a new partition to execute.

The application model we have presented in this section illustrates how avionics applications can be described in a modular way, using the synchronous language SIGNAL. A great advantage of SIGNAL-based modeling is the possibility to formally analyze descriptions. In particular, timing issues can be addressed using the performance evaluation technique implemented in POLYCHRONY.

[6] UML Specification version 2.0: Superstructure – *Object Management Group* (`www.omg.org`).

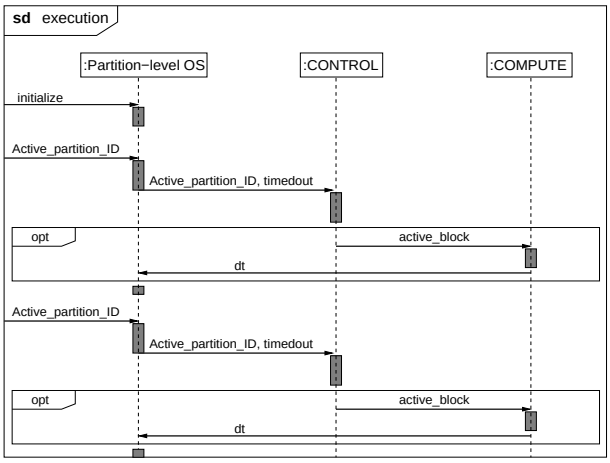


Fig. 10. A sketch of the model execution.

4.3 Model analysis: temporal evaluation

The temporal analysis of the application SATMAINT exposed in this section is based on a technique presented in [12]. It relies on the principle introduced in section 3, which consists of using an observer program to check properties of a given program.

Overview of the technique. As a general observation, a SIGNAL program is recursively composed of sub-processes, where elementary sub-processes are primitive constructs, called *atomic nodes*. A transformation of such a program substitutes each of its signals x with a new signal representing availability dates $date\_x$, automatically replacing atomic nodes with their temporal model counter-part. The resulting time model is composed with the original functional description of the application (using the standard synchronous composition). Each signal x has the same clock as its associated date information $date\_x$. The simulation of the resulting program reflects both functional and timing aspects of the original program. Obviously, a less strict temporal model can be designed in order to get faster simulation (or formal verification). It is sufficient to consider more abstract representations either of the program or of its temporal model.

The temporal interpretations of SIGNAL primitive constructs are collected in a *library of parameterized cost functions*. For a program to be interpreted, the library is extended with interpretations of external function calls and other separately compiled processes, which appear in the program. As an illustration, let us consider the following primitive construct: `z := x + y`. It is represented by the atomic node depicted by Figure 11, on the left hand side. Besides the input values `x` and `y`, this node also requires a clock information, denoted by the signal `clk_z`, which triggers the computation of the output value `z`. The associated temporal model is represented by the node illustrated on the right

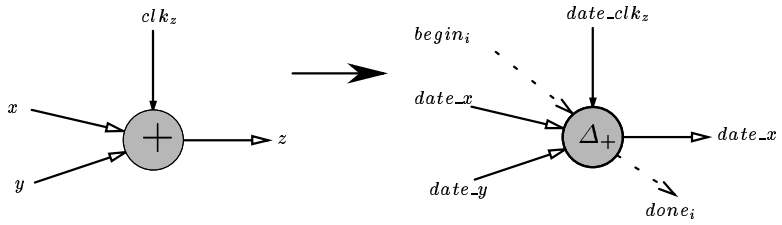


Fig. 11. Node associated with `z := x + y` (left); and its temporal model (right).

hand side. The SIGNAL program corresponding to this temporal model, called `T_Plus`, is depicted by Figure 12. In this model, `MAXn` denotes a SIGNAL process that returns the maximum value of `n` inputs among those that are present at a given instant (i.e., inputs are not constrained to be simultaneously present). The notations `type_x` and `type_y` represent respectively the types of `x` and `y`. The input signal `date_clk_z` is associated with the trigger clock `clk_z`. Signals `begin_i` and `done_i` have been added in order to express the end of execution of a given node so that the following nodes could be executed. The presence of `begin_i` in a node means that the preceding ones (following the scheduling order chosen for node execution), have already produced all their output dates. The presence of `done_i` means that the current node has calculated all its output dates (i.e., `done_i` becomes `begin_i+1`). The date of `z`, denoted by `date_z`, is the sum of the maximum date of inputs and the delay of the addition operation, some Δ_+. This quantity Δ_+ depends on the desired implementation, on a specific platform. It has to be provided in some way by the user, with respect to the considered platform. In the current implementation in POLYCHRONY, the value Δ_+ is provided by a function `DELTA_ADD` which has the types of the operands as parameters and which fetches the required value from some table. Following the same idea, each primitive construct of SIGNAL has been associated with its temporal interpretation. Thanks to compositionality of SIGNAL specifications, the same principle can be applied at any level of granularity.

```
process T_Plus{ type_x, type_y; }
 ( ? date_type date_x, date_y, date_clk_z, begin_i;
   ! date_type date_z, done_i; )
(| date_z := MAX2( MAX3( date_x, date_y, date_clk_z),
                   begin_i when ^date_z) + DELTA_ADD{type_x, type_y}()
 | done_i := (date_z default begin_i) cell ^done_i
 |);
```

Fig. 12. A temporal interpretation of `z := x + y` in SIGNAL.

In addition to the library of cost functions of primitive constructs, the implementation also requires *platform-dependent information* (e.g. the delay of the addition of two integer numbers on a given processor). For instance, the sum of integer numbers coded on 32 bits will take one cycle on a processor with a 32-bit adder (this is the case of the Intel Pentium IV processor). On the other hand, the same operation requires more than one cycle on a processor with only a 16-bit adder (like the Intel 8086 processor). In the example exposed in Figure 12, this information is obtained via the `DELTA_ADD` call.

Application to the partition SATMAINT. The *co-simulation* schema is shown in Figure 13. The model of SATMAINT is composed with its associated temporal interpretation *T_SATMAINT* (the prefix notation "T_" is used for temporal interpretation). At each simulation step, the date of an output $date(O_j)$ depends on the date of an input $date(I_i)$ and the *control configuration* represented by a "valuation" of boolean signals vector $[c_1, \ldots, c_q]$ computed in the original program. In fact, the control parts of SATMAINT and T_SATMAINT are identical, only data parts functionally differ: the data part of SATMAINT computes the functional results whereas the data part of T_SATMAINT yields date informations. The vector $[c_1, \ldots, c_q]$ contains intermediate boolean signals computed by the data part of SATMAINT that are needed by T_SATMAINT to compute output information. Note that in a straightforward approach, it is possible to provide a set of vectors that covers all the possible combinations for the control flow. A better way is to take into account the existing relationships between these booleans such as provided by the clock calculus of SIGNAL (this is expressed through the composition of the original program and its temporal interpretation).

In practice, we mainly raise one difficulty about the implementation of the schema illustrated in Figure 13. It is due to the scalability issue which can become problematic during the compiling of large application programs (here, the program resulting from the composition of the application model together with its temporal interpretation, in other words, `(| SATMAINT | T_SATMAINT |)`, is huge and may not facilitate its compiling). So, the adopted solution consists of using a modular evaluation schema, which is presented in the following. One can also mention other advantages of modularity like the possibility of applying a component-based approach, then taking advantage of re-usability.

The modularity of the SIGNAL language allows to construct a program from other programs by composition. Therefore, this principle also applies to the construction of the temporal interpretation of the partition SATMAINT, which is quite large. For that, we consider a splitting of the corresponding SIGNAL model into subparts of a reasonable size (e.g. such a subpart could be a process). They will be easier to be first addressed. So, for each subpart, we define an associated temporal interpretation. Afterwards, the resulting model can be composed with the concerned subpart. The program obtained from this composition is abstracted in order to take into account only information that are relevant for the considered observation (as a result, abstractions contribute to reduce the

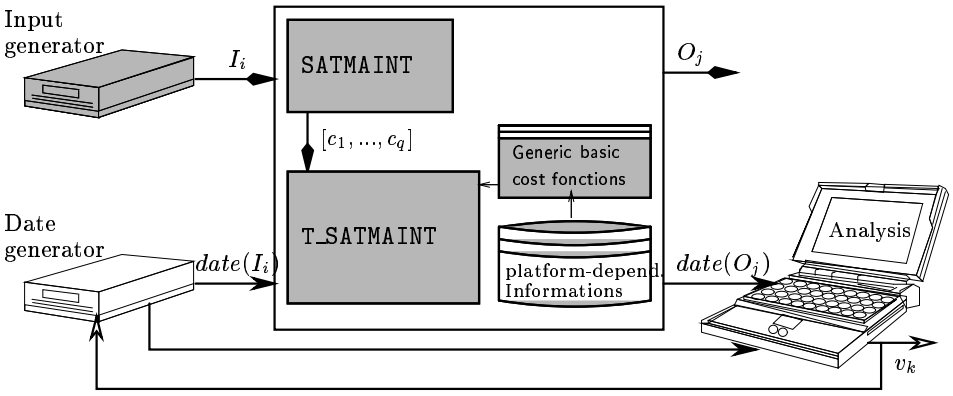


Fig. 13. Co-simulation of SATMAINT with its temporal interpretation.

size of a program). Finally, the global model that consists of the composition of the application with its temporal interpretation is obtained by composing the abstracted subprograms.

Let $\mathtt{P} \equiv \mathtt{P}_1 \mid \ldots \mid \mathtt{P}_n$ denote the program corresponding to the application that is considered for temporal analysis. We want to define the program $\mathtt{P'} \equiv (\mid \mathtt{P} \mid \mathtt{T\_P} \mid)$ that will be used for the simulation, where T_P is the temporal model of P. The same program can be also rewritten as: $\mathtt{P'} \equiv \mathtt{P'}_1 \mid \ldots \mid \mathtt{P'}_n$. Each $\mathtt{P'}_i$ denotes the composition of a subprogram $\mathtt{P}_i$ of P, with its associated temporal interpretation $\mathtt{T\_P}_i$. The following steps are identified in order to carry out experiments:

1. *Partial definition of temporal interpretations*: for each subprogram $\mathtt{P}_i$ ($i \in \{1, ..., n\}$) of P, we define the corresponding temporal model $\mathtt{T\_P}_i$.
2. *Composition of each subpart of the application with its associated temporal interpretation, then abstraction*: this step defines the subprograms $\mathtt{P'}_i$ ($i \in \{1, ..., n\}$), which constitute the simulation program P' that we want to construct. The abstraction aims to keep only relevant information of these subprograms for the co-simulation. It follows that $\mathtt{P'}_i \equiv \alpha(\mathtt{P}_i \mid \mathtt{T\_P}_i)$, where α denotes the abstraction (e.g. a program can be abstracted by considering only its control part—boolean and synchronization signals—or by approximating the value of numerical signals—for instance, by dealing with domains of intervals instead of point-wise domains).
3. *Construction of the global model for simulation*: this model results from the composition of subprograms $\mathtt{P'}_i$, defined at the previous step, i.e. $\mathtt{P'} \equiv \mathtt{P'}_1 \mid \ldots \mid \mathtt{P'}_n$.

On the other hand, we notice that the above method can be applied in a recursive way. In a program P, equivalent to $\mathtt{P}_1 \mid \ldots \mid \mathtt{P}_n$, when the size of the subprograms $\mathtt{P}_i$ ($i \in \{1, ..., n\}$) is important, we can also use the same method

for each one in order to define the corresponding P'$_i$ ($i \in \{1, ..., n\}$). The global simulation model then results from the composition of P'$_i$s.

To apply the method to the partition SATMAINT, we should first decompose it into subparts. Let us consider processes as subparts of the partition. For instance, in the process PROC_8, we can begin by defining the interpretation of its COMPUTE subpart: T_COMPUTE $\equiv$ T_BLOCK_0 | ... | T_BLOCK_7. In this composition, we suppose that the temporal model of each block is obtained without necessarily having to consider its decomposition as is done for subprograms of a larger size. The temporal interpretation of the CONTROL subpart of the process is determined in a similar way as the COMPUTE one. The composition of both temporal models gives the model for PROC_8: T_PROC_8 $\equiv$ T_COMPUTE_8 | T_CONTROL_8. We proceed in the same way for the other subparts of the partition. Then, we compose the resulting temporal model with its associated program subpart. For PROC_8, it follows: (| PROC_8 | T_PROC_8 |).

This process could be now "simplified" by considering approximations (or abstractions). This will facilitate the compiling of the global program. For instance, let us consider a subpart of the application that performs a complex operation, which requires a constant duration δ on a target platform (by "complex", we mean an operation that requires several elementary operations, e.g. product of two matrices). The temporal model of such a subpart can be defined in a simple way by adding the constant δ to the dates corresponding to inputs availability in order to compute dates associated with outputs. This interpretation is simpler than the one obtained by composing the temporal models of all intermediate operations that are carried out by the considered subprogram. Other simplifications consist of considering worst case execution times [17] in the definition of the temporal interpretation of a subprogram. Finally, interesting possible abstractions of subparts of the application can be obtained by considering only their control parts. They provide enough information for the co-simulation with the associated temporal models.

5 Discussion and related work

We observe that modularity and abstraction play a central role in SIGNAL programming for scalability in our design approach. Basically, the description of a large application is achieved with respect to a precise design methodology that consists of specifying first, either completely or partially (by using abstractions), sub-parts of the application. After that, the resulting programs can be composed in order to obtain new components. These components can be also composed and so on, until the application description is complete. The construction of the global simulation model of SATMAINT for temporal issues relies on this principle. The description of the application itself relies on the same principle. A crucial issue about the design of safety critical systems, like in avionics, is the correctness of these systems. In POLYCHRONY, the functional properties of a system can be checked using tools like the compiler or the model checker SIGALI. Here, we addressed temporal aspects of programs. For that, we used a

technique consisting of co-simulating the program under analysis with an associated observer (also referred to as temporal interpretation) defined in SIGNAL. The observer is another program which has the same control as the observed one, but its data part reflects the temporal dimension of the analyzed program. Using SIGNAL for both the model of an application and its associated temporal interpretation results in uniform descriptions upon which available tools and techniques remain applicable.

We can mention a few studies addressing the design of embedded real-time systems in the avionic domain. The first one is the COTRE approach [5]. Its main objective consists in providing the designer with a methodology, an Architecture Description Language (ADL) called *Cotre*, and an environment to describe, verify and implement embedded avionic systems. The Cotre language distinguishes two different views for descriptions: a user view expressed using the *Cotre for User* language (termed *U-Cotre*) and a view for verification (termed *V-Cotre*). In fact, the latter plays the role of an intermediate language between U-Cotre and certain existing verification formalisms (e.g. timed automata, timed Petri nets). The authors argue that the use of formal techniques is one of the main differences between the Cotre language and other ADLs. COTRE is closely related to the approach based on the *Avionics Architecture Description Language* (AADL), which is developed by the International Society of Automotive Engineers (SAE) [8]. It is dedicated to the design of the software and hardware components of an avionics system and the interfaces between those components. The AADL definition is based on METAH (an ADL developed by Honeywell) [22]. It allows to describe the structure of an embedded system as an assembly of software and hardware components in a similar way. The AADL draft standard also includes a UML profile of the AADL. This enables the access to formal analysis and code generation tools through UML graphical specifications.

While these approaches combine various formalisms and tools for the design of embedded real-time systems, our approach relies on the single semantic model of the SIGNAL language. It is very important to have a common framework in order to guarantee the correctness of the designs. Modularity allows to overcome scalability problems. In addition, such a framework favors MDA-like approach since all transformations of descriptions are achieved with respect to a unique semantic model.

Among specific studies related to IMA, we can mention those concerning the two-level hierarchical scheduling aspects within IMA systems. In [3], Audsley and Wellings introduced a scheduling approach for APEX applications. In [14], Lee et *al.* presented algorithms in order to produce cyclic partition and channel schedules for IMA-based avionics systems. The technique we illustrated in this paper for temporal analysis provides information on execution times of partitions. Thus, these information could be used when taking decisions in processor allocation to partitions. Further expected benefits of defining our approach in a formal framework are the available techniques and tools that help to address some critical issues of IMA such as the partitioning, which still need to be further explored by researchers. Indeed, in current industrial practices, avionic functions

with high critical level are designed using federated architectures (for instance, this is the case for the future Airbus A380). This is likely due to the fact that partitioning raises several questions that are not enough addressed yet. Among these questions, we can mention the correctness of a partitioning, which is crucial. A formal description of partitioning requirements is proposed by Di Vito [7], using the language of PVS (Prototype Verification System). However, this description only concerns space partitioning (time partitioning is not addressed). The use of a dataflow representation such as in SIGNAL can allow to define a correct-by-construction partitioning, based on a so-called *sensitivity analysis* [10]. Being able to guarantee the correctness of a given partitioning can help reducing IMA certification efforts. A study addressing this last issue has been done by Conmy and McDermid [6], who propose a high level failure analysis of IMA. The analysis is part of an overall IMA certification strategy. Finally, a presentation of the IMA-based communication network designed for the future Airbus A380 is given by Sánchez-Puebla and Carretero in [20].

6 Conclusions

The work presented in this paper promotes modular designs in a formal context. It favors component based designs (where an immediate benefit is re-usability), and verification and validation. This has been possible through the use of the synchronous language SIGNAL. We showed how a synchronous model of a real world avionics application is described following the integrated modular concept (IMA). The description relies on an existing library of SIGNAL models of APEX services defined by the avionics standard ARINC 653. Issues on the temporal evaluation of the defined model have been also addressed in order to allow the analysis of real-time behaviors of the application. By considering a unique formalism (i.e SIGNAL) and its associated tool-set (i.e POLYCHRONY) for both description and analysis of applications, we put forward a strongly uniform formal design approach, which allows to reason about the properties of the applications.

References

1. Airlines Electronic Engineering Committee. ARINC Report 651-1: Design Guidance for Integrated Modular Avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, November 1997.
2. Airlines Electronic Engineering Committee. ARINC Specification 653: Avionics Application Software Standard Interface. In *Aeronautical radio, Inc., Annapolis, Maryland*, January 1997.
3. N.C. Audsley and A.J. Wellings. Analysing APEX Applications. In *Proceedings of Real Time Systems Symposium (RTSS'96)*, 1996.
4. A. Benveniste *et al.* The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
5. B. Berthomieu *et al.* Towards the Verification of Real-Time Systems in Avionics: the Cotre Approach. In *Electronic Notes in Theoretical Computer Science, vol. 80*, 2003.

6. P. Conmy and J. McDermid. High Level Failure Analysis for Integrated Modular Avionics. In *Proceedings of the 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Brisbane, Australia*, June 2001.
7. B.L. Di Vito. A Model of Cooperative Noninterference for Integrated Modular Avionics. In *Proceedings of Dependable Computing for Critical Applications (DCCA-7), San Jose, CA*, January 1999.
8. P.H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *Proceedings of Workshop on Model-Driven Embedded Systems, Washington D.C., USA*, May 2003.
9. A. Gamatié and T. Gautier. Synchronous Modeling of Avionics Applications using the SIGNAL Language. In *Proceedings of 9th IEEE Real-time/Embedded Technology and Applications Symposium*. Washington D.C., USA, May 2003.
10. T. Gautier and P. Le Guernic. Code generation in the sacres project. In *Proceedings of the Safety-critical Systems Symposium, SSS'99, Springer*. Huntingdon, UK, February 1999.
11. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST'93), Springer Verlag, Twente*, June 1993.
12. A. Kountouris and P. Le Guernic. Profiling of SIGNAL Programs and its Application in the Timing Evaluation of Design Implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9. HP Labs, Bristol, February 1996.
13. P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design, World Scientific*, 12(3):261–303, June 2003.
14. Y.-H. Lee *et al.* Resource Scheduling in Dependable Integrated Modular Avionics. In *Proceedings of the International Conference on Dependable Systems and Networks*, April 2000.
15. F. Maraninchi and Y. Rémond. Mode-Automata: About Modes and States for Reactive Systems. In *Proceedings of the European Symposium On Programming, Lisbon, Portugal, Springer-Verlag*, pages 39–44, March 1998.
16. H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the SIGNAL Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.
17. P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.
18. J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. In *NASA Contractor Report 4673, num. SRI-CSL-95-1*, Menlo Park, CA, March 1995.
19. J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Technical report, NASA Langley Research Center, June 1999.
20. M.A. Sánchez-Puebla and J. Carretero. A new Approach for Distributed Computing in Avionics Systems. In *Proceedings of the 1st International Symposium on Information and Communication Technologies, Dublin, Ireland*, 2003.
21. J. Sifakis. Modeling Real-Time Systems - Challenges and Work Directions. In *Proceedings of the First International Workshop on Embedded Software, T.A. Henzinger and C.M. Kirsch, Eds, LNCS 2211, Springer Verlag*, October 2001.
22. S. Vestal. METAH Support for Real-Time Multi-Processor Avionics. In *Proceedings of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1997.